# Rust Book - 10.1: Generic Types

#rust Generic Types, Traits, and Lifetimes

Every programming language has tools for effectively handling the duplication of concepts. In Rust, one such tool is*generics*. Generics are abstract stand-ins for concrete types or other properties. When we're writing code, we can express the behavior of generics or how they relate to other generics without knowing what will be in their place when compiling and running the code.

Similar to the way a function takes parameters with unknown values to run the same code on multiple concrete values, functions can take parameters of some generic type instead of a concrete type, like `i32` or `String`.

## Removing Duplication by Extracting a Function

```rust
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let mut largest = number_list[0];
    for number in number_list {
        if number > largest {
            largest = number;
        }
    }
    println!("The largest number is {}", largest);
}
```

```rust
fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

## Generic Data Types

### Generic Data Types

We can use generics to create definitions for items like function signatures or structs, which we can then use with many different concrete data types.

### In Function Definitions

```

    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

Compile error:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^
  |
  = note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

## In Struct Definitions

```
struct Point<T> {
    x: T, y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
    // FAILS!
    // expected integral variable, found floating-point variable
    let wont_work = Point { x: 5, y: 4.0 };
}
```

To define a `Point` struct where `x` and `y` are both generics but could have different types, we can use multiple generic type parameters. We can change the definition of `Point` to be generic over types `T` and `U` where `x` is of type `T` and `y` is of type `U`.

```
struct Point<T, U> {
    x: T, y: U,
}

// WORKS!
let both_integer = Point { x: 5, y: 10 };
let both_float = Point { x: 1.0, y: 4.0 };
let integer_and_float = Point { x: 5, y: 4.0 };
```

You can use as many generic type parameters in a definition as you want, but using more than a few makes your code hard to read. When you need lots of generic types in your code, it could indicate that your code needs restructuring into smaller pieces.

## In Enum Definitions

```
enum Option<T> {
    Some(T),
    None,
}
```

`Option<T>` is an enum that is generic over type `T` and has two variants: `Some`, which holds one value of type `T`, and a `None` variant that doesn't hold any value. By using the `Option<T>` enum, we can express the abstract concept of having an optional value, and because `Option<T>` is generic, we can use this abstraction no matter what the type of the optional value is.

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

## In Method Definitions

```
struct Point<T> {
    x: T, y: T,
}
impl<T> Point<T> {
    fn x(&self) -> &T { &self.x }
}
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10 };
    println!("p.x = {}", p1.x());

    // COMPILE ERROR!
    // WONT WORK with i32 Point!
    // method `distance_from_origin` not found
    println!("p.dist: {}", p1.distance_from_origin());

    let p2 = Point { x: 5.5, y: 10.1 };
    // WORKS with f32 Point!
    println!("p.dist: {}", p2.distance_from_origin());
}
```

Generic type parameters in a struct definition aren't always the same as those you use in that struct's method signatures. The method `mixup` takes another `Point` as a parameter, which might have different types than the `self` `Point` we're calling `mixup` on. The method creates a new `Point` instance with the `x` value from the `self` `Point` (of type `T`) and the `y` value from the passed-in `Point` (of type `W`).

```
impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c'};
    let p3 = p1.mixup(p2);
    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

## Performance of Code Using Generics

Performance of Code Using Generics

Rust implements generics in such a way that your code doesn’t run any slower using generic types than it would with concrete types.

Rust accomplishes this by performing **monomorphization** of the code that is using generics at compile time. Monomorphization is the process of turning generic code into specific code by filling in the concrete types that are used when compiled.

In this process, the compiler does the opposite of the steps we used to create the generic function: the compiler looks at all the places where generic code is called and generates code for the concrete types the generic code is called with.

```
let integer = Some(5);
let float = Some(5.0);
```

When Rust compiles this code, it performs monomorphization. During that process, the compiler reads the values that have been used inOption<T>instances and identifies two kinds ofOption<T>: one isi32and the other isf64. As such, it expands the generic definition ofOption<T>intoOption_i32andOption_f64, thereby replacing the generic definition with the specific ones.

The monomorphized version of the code looks like the following. The genericOption<T>is replaced with the specific definitions created by the compiler:

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

Because Rust compiles generic code into code that specifies the type in each instance, we pay no runtime cost for using generics. When the code runs, it performs just as it would if we had duplicated each definition by hand. The process of monomorphization makes Rust’s generics extremely efficient at runtime.